TR 01-020

Dynamic Load Balancing Algorithms for Sequence Mining

Valerie Guralnik and George Karypis

May 08, 2001

Dynamic Load Balancing Algorithms for Sequence Mining [*]

Valerie Guralnik, George Karypis
{guralnik, karypis }@cs.umn.edu
Department of Computer Science and Engineering/Army HPC Research Center
University of Minnesota

May 4, 2001

Abstract

Discovery of sequential patterns is becoming increasingly useful and essential in many scientific and commercial domains. Enormous sizes of available datasets and possibly large number of mined patterns demand efficient and scalable algorithms. In this paper we present a parallel formulation of a serial sequential pattern discovery algorithm based on tree projection that uses a novel dynamic load balancing algorithm which is well suited for distributed memory parallel computers. Our experimental evaluation on a 32 processor IBM SP show that this algorithms are capable of achieving good speedups, substantially reducing the amount of the required work to find sequential patterns in large databases.

1 Introduction

Sequence data arises naturally in many applications. For example, marketing and sales data collected over a period of time provide sequences that can be analyzed and used for projections and forecasting. In the past several years there has been an increased interest in using data mining techniques to extract interesting sequential patterns from temporal sequences. The most time consuming operation in the discovery process of such patterns is the computation of the frequency of the occurrences of interesting sub-sequences of set of events (called candidate patterns) in the database of sequences. However, the number of sequential patterns grows exponentially and various formulations have been developed [AS96, MTV95, SA96, JKK99] that try to contain the complexity by imposing various temporal constraints, and by consider only those candidates that have a user specified minimum support. Even with these constraints, the task of finding all sequential patterns requires a lot of computational resources (*i.e.*, time and memory), making it an ideal candidate for parallel processing. This was recognized by Zaki [Zak99] which developed a parallel formulation of the SPADE algorithm [Zak98] for shared-memory parallel computers.

In [GGK01], we proposed two different parallel algorithms for finding sequential association rules on distributed-memory parallel computers. The first algorithm decomposes the computation by exploiting data parallelism, whereas the other utilizes task parallelism. Experimental evaluation showed that the algorithms incur small communication overheads, achieve good speedups, and can effectively utilize the different processors, and that static task-parallel algorithm outperformed data-parallel algorithm. However, as number of processors increased, the accuracy of estimated work-load decreased and the computation became increasingly un-balanced.

To overcome this problem, in this paper we developed a different parallel algorithm for finding sequential association rules on distributed-memory parallel computers that utilizes task parallelism along with dynamic load balancing. One of the key contributions of this paper is development of dynamic scheme that minimizes idle time in case when distributed workload is unbalanced. We experimentally evaluated the performance of proposed algorithm on different datasets on a 32-processor IBM SP2 parallel computer. Our experiments show that the proposed algorithm achieves good speedups and outperforms static load balancing scheme proposed in [GGK01].

[*]This work was supported by NSF CCR-9972519, EIA-9986042, ACI-9982274, by Army Research Office contract DA/DAAG55-98-1-0441, by the DOE ASCI program, and by Army High Performance Computing Research Center contract number DAAH04-95-C-0008. Access to computing facilities was provided by the Minnesota Supercomputing Institute.

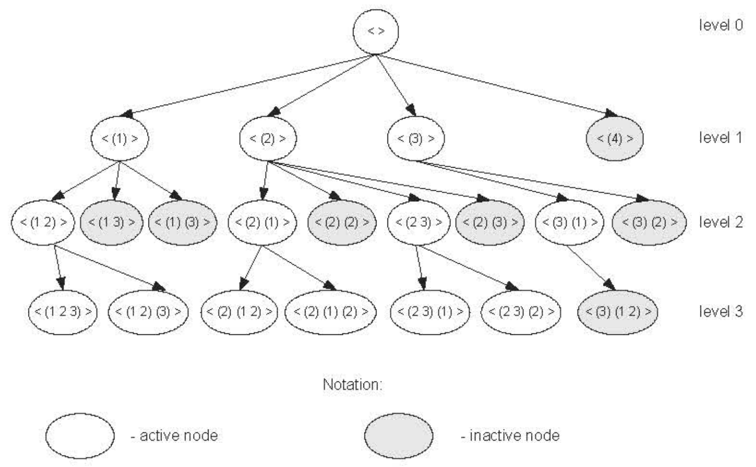

Figure 1: An example of Projection Tree

2 Sequence Mining

The problem of mining for sequential patterns was first introduced by Agrawal et al [AS96]. The authors showed how their association rule algorithm for unordered data [AS94] could be adapted to mine for frequent sequential patterns in sequence data. The class of episodes being mined was generalized, and the performance enhancements were presented in [SA96]. In this section we will summarize the terminology first introduced by Agrawal [AS96] and being used throughout the paper.

We are given a database $\mathcal{D}$ of sequences called *data-sequences*. Each data-sequence consists of the list of *transactions*, ordered by increasing transaction-time. A transaction has the following fields: sequence-id, transaction-id, transaction-time, and the items present in the transaction. We assume that the set of items $\mathcal{I} = \{i_1, i_2, \ldots, i_m\}$, is the set of literals that can be sorted in lexicographical order. The items in the transaction are sorted in lexicographical order.

An **itemset** i is a non-empty set of items, denoted by $(i_1 i_2 \ldots i_m)$, where i_j is an item. A **sequence** is an ordered list of itemsets, denoted by $< s_1 s_2 \ldots s_n >$, where s_i is an itemset. The **support of a sequence** is defined as the fraction of total data-sequences that contain this sequence. A sequence is said to be **frequent** if its support is above a certain user-specified minimum threshold. Given a database $\mathcal{D}$ of data-sequences, the problem of mining for sequential patterns is to find the *all frequent sequences* among all data-sequences. Each such frequent sequence represents a *sequential pattern*. It is important to note that from now on the term sequential is adjective of pattern, while term serial is adjective of algorithm.

2.1 Serial Tree Projection Algorithm

The tree projection algorithm represents discovered patterns in the tree structure, called Projection Tree (PT). The tree is grown progressively by the algorithm such that only the nodes corresponding to frequent patterns are generated. The level-wise version of the algorithm grows the tree in a breadth-first manner. In iteration k, it extends all the nodes at level $k-1$. The candidate extensions of a given node are formed by using only the frequent extensions of its parent. A node can be extended into multiple children nodes via items in two ways as follows. A child of the node can be created by extending the last itemset in a pattern represented by a node with an item that is lexicographically larger than all the items in that itemset (called *itemset extension*) or by extending a pattern with a new 1-item itemset (called *sequence extension*). The new children represent $(k+1)$-item patterns. All the nodes that belong to a sub-tree which potentially can be extended are called *active extensions*. If the node cannot be extended any further it becomes *inactive* and is pruned from the tree.

Figure 1 illustrates an example of Projection Tree. In this example the set of active extensions of node $< (2) >$ is $\{1,\ 3\}$, where 1 is an active sequence extension and 3 is an active itemset extension. The set of active items is $\{1,\ 2,\ 3\}$.

One of the key features of the algorithm is that the supports of the patterns represented by the candidate extensions are gathered by using the set of *projected* data-sequences at the parent. The algorithm maintains a list of *active items* of the node. Active item list of a node consists of items that can be found in itemset represented by its descendants. When a data-sequence is projected on a node, only the items that occur in its active item list are kept. The data-sequence gets recursively projected along the paths determined by active extensions. The idea is, only those items in a data-sequence percolate down the tree that can only potentially be useful in extending the tree by one more level.

The process of counting support of $(k+1)$-item pattern is accomplished as follows. Each node, representing the pattern $P = < s_1 s_2 \ldots s_m >$, at level $k-1$ maintains four count matrices, which are used to count the support of four different types of patterns. Once data-sequences are projected to the node with matrices, the algorithm iterates through them to gather support counts. For more detail refer to [GGK01].

3 Recent Research

We recently developed two different approaches for parallelizing the sequential tree projection algorithm, both of which use static assignment of work to processors [GGK01].

The first approach exploits the data parallelism that exists in computing the support at each node. In particular, it works as follows. If p is the total number of processors, the original database is initially partitioned into p equal size parts, and each one is assigned to a different processor. To compute the support of the candidate sequences at level k, each processor projects its local set of data sequences to the nodes at level $k-2$, and computes their support based on the local data sequences. The global supports are determined by using a reduction operation to add up the individual supports. These global supports are made known to all the processors, and are used to determine which nodes at the kth level meet the minimum support constraints. Note that in this approach, all the processors build the same tree (which is identical to that built by the serial algorithm). Even though this approach does lead to load balanced computations it incurs high communication overhead.

The second approach exploits the task parallelism that exists in the tree-based nature of the computation. The key idea behind the task parallel formulation (STPF) is that when the support of the candidate patterns at level k is computed by projecting the databases at the various nodes at the $k-2$ level of the tree, the computations at each of these $k-2$ nodes are independent of each other. Thus, the computations at each node becomes an independent task and the overall computation can be parallelized by distributing these tasks among the available processors. The task parallel formulation distributes the tasks among the processors in the following way. First, the tree is expanded using the data-parallel algorithm described above, up to a certain level $k+1$, with $k > 0$. Then, the different nodes at level k are distributed among the processors. Once this initial distribution is done, each processor proceeds to generate the subtrees (*i.e.*, sub-forest) underneath the nodes that it has been assigned.

The key step in the STPF algorithm is the method used to partition the nodes of the kth level of the tree into p disjoint sets $S_0, S_1, \ldots, S_{p-1}$. To ensure load balance and minimize the database overlap we use minimum-cut bipartite graph partitioning approach described in [GGK01]. The essence of this approach is to assign to each processor nodes of the tree that need access to the same set of data-sequences as much as possible.

4 Dynamic Load Balancing

The underlying assumption for the efficient execution of STPF algorithm described in Section 3 is that the relative amount of work required to expand each one of the tree nodes can be accurately estimated beforehand. The experiments, presented in [GGK01] and in Section 5, show that even though estimates are in general accurate, their accuracy tends to decrease as the number of processors increases. Moreover, there exist data sets for which the estimates can be substantially wrong, even for small number of processors. For this reason we developed a dynamic task parallel formulation (DTPF) that monitors the load imbalance and moves work between the processors, as needed.

The topic of dynamic load balancing of tree-based computations has been extensively studied in the context of parallel heuristic state-space algorithms [KGGK94] and a number of different approaches have been developed. A simple way to load balance the computation is do that in a synchronous fashion [KGGK94, KK94]. In this approach after the nodes and database are partitioned between processors, the processors synchronously extend the tree level by level. After each level expansion the processors communicate between each other to determine whether the work needs to be re-balanced. If the conclusion is made that workload re-distribution is necessary, then the processor P_i with lth maximum estimated workload W_i sends work to the processor P_j with lth minimum estimated workload W_j. The portion of the work to be send to the receiving processor can be determined randomly or the minimum cut bipartite graph partitioning approach described in [GGK01] cab be used to select the parts of the tree that lead to the least amount of data sharing overhead.

Unfortunately, even though this approach is quite simple to implement, it does not significantly reduce the load imbalance overhead. This is primary due to the fact that number of possible load balancing points is quite small (equal to the depth of the tree). To see this, consider a scenario in which a computation at certain level is particularly unbalanced. The imbalance will only be determined after the counting of candidate extensions has been completed at that level. At this point nothing can be done to correct this imbalance. If this imbalanced computation contributes to the majority of the parallel imbalance, the synchronous dynamic load balancing scheme cannot do anything to correct it. Our experiments with this approach (not presented here) verified this observation, as we were not able to substantially reduce the load imbalance overhead.

For this reason we focused on developing dynamic load balancing algorithm that uses asynchronous paradigms, that are described in the next section.

4.1 Asynchronous Dynamic Task Parallel Formulation

Our asynchronous dynamic load balancing algorithm is similar in nature to the task parallel formulation algorithm described in Section 3. That is, the tree is extended to level $k+1$ using the data parallel formulation and then the bipartite graph partitioning algorithm is used to partition the nodes at level k among the p processors. However, instead of allowing each processor to expand its different subtrees independently to the very end, the processors check to determine whether the work needs to be re-balanced.

To accomplish this, one can use receiver initiated load balancing with random pooling scheme [KGGK94]. In this scheme, as a processor becomes idle (that is it finishes its portion of allocated work), it randomly selects a donor processor and sends it a work request. The donor sends a response indicating whether or not it has additional work. If a response indicates that a donor doesn't have anymore work, the processor selects another donor and sends work request to that donor. Otherwise, the processor receives nodes to expand, along with the portion of the database to associated with those nodes. Upon receiving new work the processor starts extending newly received nodes. This process continues until every processor completely extended the nodes assigned to it. Dijkstra's token termination algorithm [DSG83] can be used to detect whether all processors have become idle.

The key issue in this approach is when will the processors service work requests. The natural point at which an active processor can service a request is the time between the point it completed counting support of candidate extensions at a certain level and is ready to count support of candidate extensions at the next level. This is because, this is the time at which work can be transferred by sending only some of the nodes of the tree along with relevant portions of the locally stored databases. On the other hand, if we allow work transfered to take place while a processor is in the middle of counting the support of the candidate patterns at the next level, then besides the tree nodes and their respective portions of the database, we also need to send the partial counting matrices associated with the different nodes. This will substantially increase the amount of data that needs to be transferred, severely reducing the gains achieved by dynamic load balancing.

However, the problem associated with servicing work requests only at the natural breaking points of the computation is the following. The bulk of the computation is performed during the counting phases, as a result if processor P_i sends a work request to processor P_j right after P_j has started its counting phase, it will take a significant amount of time before P_j services that request. Through out that time processor P_i will remain blocked, waiting for work. One way of eliminating this idling is to follow a protocol in which processor P_j periodically checks for request (while working on its counting phase), and responds to them by indicating that even though it has work it cannot send it right away. Upon receiving such a response, processor P_i can either decide to wait for P_j or ask another processor for work.

This new load balancing protocol can eliminate of the waiting time incurred in receiving work. However,

Dataset	Avg. no. of transactions per sequence	Avg. no. of items per transaction	Avg. length of maximal potentially frequent sequences	Avg. sie of itemsets maximal potentially frequent sequences	$MinSup(\%)$
DS1	10	2.5	4	1.25	0.1
DS2	10	5	4	1.25	0.25
DS3	10	5	4	2.5	0.33
DS4	20	2.5	4	1.25	0.25

Table 1: Parameter values for datasets

as discussed earlier, because the depth of the trees are quite small, the overall number of natural work transfer points is quite small. As a result, a processor can still be blocked for a long time. To address this problem we developed the following algorithm. Instead of servicing work request only during the natural points, we follow a policy in which if a request comes during the early stages of each counting phase, then the processor can discard the work it has done so far and service the work request. Because the amount of work done so far is not substantial, this will not affect the overall time significantly. Instead the computation will benefit from the load balancing.

The overall structure of the developed algorithm is the following. If a work request comes at the early stages of a counting phase, an active processor discards work done so far and shares its workload with the recipient. If a work request comes at the middle stages of the counting phase, an active processor lets the receiver know that while it has work, the work is not available immediately. Along with this message, the active processor also sends an estimated time before completion as well as an estimated relative amount of workload at the next level. Towards the end of the counting phase, an active processor ignores the work requests. The requests received during that time are processed upon completion of the counting phase at that level.

If an idle processor doesn't receive work from a potential donor (either because the donor doesn't have work or in the middle of its counting phase), it chooses another processor to inquire about work. If after polling all available processors, an idle processor didn't get any work, this processor picks an active processor with work and lets it know that it will wait for work until it gets it. To ensure that idle processors will pick different active processors, an idle processor will pick top n (in terms of estimate of waiting time and remaining workload) active processors and will randomly choose one to attach itself to.

5 Ep erimental Evaluation

5.1 Experimental Setup

We use the same synthetic datasets as in [SA96], albeit with more data-sequences. We generated datasets by setting number of maximally potentially frequent sequences $N_S = 5000$, number of maximal potentially frequent itemsets $N_I = 25000$ and number of items $N = 10000$. The number of data-sequences D was set to 1 million. Table 1 summarizes the dataset parameter settings and shows the minimum support that were used in experiments reported below. Note that for each of the datasets we used different support threshold. These are the same thresholds used in experiments presented in [GGK01] and were carefully selected so that resulting tree is deep enough to parallelize computation, but number of discovered sequential patterns is such that the algorithms finishes in reasonable amount of time. We ran our experiments on IBM SP cluster. The IBM SP consists of 79 four-processor and 3 two-processor machines with 391 GB of memory. The machines utilize the 222 MHz Power3 processors.

5.2 Results

The performance of our task parallel tree projection algorithm that uses dynamic load balancing (DTPF) was compared against the best of the static load balancing algorithms described in [GGK01], namely the task parallel formulation (STPF). In particular, we used both algorithms to find the frequent sequential patterns in all four datasets on 2, 4, 8, 16, and 32 processors. The amount of time required for each of these experiments is shown in Table 2. Recall from Section 4.1, that our dynamic load balancing algorithm

	DS1					DS2				
Approach	2	4	8	16	32	2	4	8	16	32
STPF	1689.95	676.83	255.21	109.29	56.7	2933.57	1224.34	445.42	241.6	119.72
DTPF-25-50-25	1746.43	643.13	236.59	111.42	60.1	3025.35	1124.99	448.24	201.9	119.01
DTPF-25-70-5	1644.66	663.66	247.36	110.99	63.36	2884.1	1239.56	477.06	215.66	122.32
DTPF-0-75-25	1635.09	648.83	242.51	107.13	62.34	2774.06	1131.99	450.78	213.89	128.17

	DS3					DS4				
Approach	2	4	8	16	32	2	4	8	16	32
STPF	2516.28	1035.96	901.38	396	314.37	2169.73	983.75	389.54	184.7	147.95
DTPF-25-50-25	2386.42	876.37	391.35	193.07	116.62	2248.32	857.14	353.23	165.47	100.92
DTPF-25-70-5	2339.03	901.79	431.8	216.12	135.43	2180.04	918.62	388.25	179.84	106.3
DTPF-0-75-25	2342.18	840.73	388.76	189.75	129.37	2183.5	828.06	344.22	185.86	107.06

Table 2: Execution Time (in secs)

DS1					DS2				
2	4	8	16	32	2	4	8	16	32
1.03	1.07	1.05	1.09	1.23	1.04	1.09	1.07	1.21	1.35

DS3					DS4				
2	4	8	16	32	2	4	8	16	32
1.13	1.31	2.83	2.32	4.11	1.005	1.03	1.11	1.23	1.22

Table 3: Load Balance of STPF approach

divides the counting phase of each processor into three stages, the early, intermediate, and late, and uses a different protocol for handling work requests in each of these stages. The results in Table 2 show three different instances of DTPF. The first instance DTPF-25-50-25 corresponds to an algorithm in which each one of the early and late stages represent approximately 25% of the overall computation (with respect to a particular counting phase), and the remaining 50% correspond to the intermediate stage. Similarly, the results labeled "DTPF-25-70-5" and "DTPF-0-75-25" correspond to instances of the algorithm with a 25%, 70%, 5%, and a 0%, 75%, 25% division between early, intermediate, and late stages, respectively.

A number of interesting observations can be made by looking at these results. First, both of the algorithms achieve super-linear speedups as we increase the number of processors from 2 to 8. This is consistent with experiments presented in [GGK01] for the STPF algorithm and is due to the fact that each processor is assigned a sub-forest of the original tree and the databases are re-distributed. As a result, the amount of the time spent in projecting and disk I/O actually reduces as the number of processors increases.

Second, comparing DTPF against STPF we can see that for each one of the different experiments DTPF achieves comparable or substantially lower run-times. In particular, focusing on the 32-processor results we can see that for DS1 and DS2, DTPF is comparable to STPF and for DS3 and DS4, DTPF-25-50-25 is 2.7 and 1.5 times faster, respectively. The dramatic improvement achieved by DTPF on the DS3 is due to the fact that the underlying tree leads to highly unbalanced computations, which the DTPF can balance by dynamically moving work between the processors. To illustrate that we calculated workload imbalance for STPF for all four datasets. These results are shown in Table 3. As we can see from that table, STPF algorithm produces highly unbalanced computations for DS3 for any number of processors. However for the remaining data sets, the overall computations are not significantly unbalanced, and for this reason the gains achieved by DTPF are much smaller, and in some cases the overhead induced by DTPF (due to its dynamic load balancing protocol and termination detection algorithm) can lead to slight increases in overall parallel execution time. For example, the computation produced by STPF approach on DS4 run on two processors had load balance equal to 1.005. As a result, the computation doesn't benefit from using dynamic load balancing, but instead suffers from overhead incurred by DTPF.

Finally, comparing the different instances of the DTPF algorithm we can see that for smaller number of processors, DTPF-0-75-25 does somewhat better compared to DTPF-25-50-25, but as number of processors increases, DTPF-25-50-25 achieves lower run-times. In particular on 32 processors, DTP-25-50-25 is about 4%-12% faster. The reason is that for small number of processors, the overall number of work transfer requests is quite small, and can be arrive within the small time window between successive counting phases.

However, as the number of processors increases, the large time window of the early stage allows the algorithm to reduce idling overheads.

6 Conclusion

In this paper we presented a new load balancing algorithm for finding sequential patterns using the tree projection algorithm that is suitable for for distributed memory parallel computers. Our experiments show that it achieves good speedups as the number of processors increase. Furthermore, the overall performance is improved in comparison to static load balancing algorithm.

References

[AS94] R. Agrawal and R. Srikant. Fast algorithms for mining association rules. In *Proc. of the 20th VLDB Conference*, pages 487–499, Santiago, Chile, 1994.

[AS96] R. Aggrawal and R. Srikant. Mining sequential patterns. In *Proc. of the Int'l Conference on Data Engineering (ICDE)*, Taipei, Taiwan, 1996.

[DSG83] E. W. Dijkstra, W. H. Seijen, and A. J. M. Van Gasteren. Derivation of a termination detection algorithm for a distributed computation. *Information Processing Letters*, 16–5:217–219, 1983.

[GGK01] Valerie Guralnik, Nivea Garg, and Vipin Kumar. Parallel tree projection algorithm for sequence mining. In *EuroPar2001*, Manchester, UK, 2001.

[JKK99] Mahesh V. Joshi, George Karypis, and Vipin Kumar. Universal formulation of sequential patterns. Technical report, University of Minnesota, Department of Computer Science, Minneapolis, 1999.

[KGGK94] Vipin Kumar, Ananth Grama, Anshul Gupta, and George Karypis. *Introduction to Parallel Computing: Design and Analysis of Algorithms.* Benjamin/Cummings Publishing Company, Redwood City, CA, 1994.

[KK94] George Karypis and Vipin Kumar. Unstructured tree search on simd parallel computers. *Journal of Parallel and Distributed Computing*, 22(3):379–391, September 1994.

[MTV95] H. Mannila, H. Toivonen, and A. I. Verkamo. Discovering frequent episodes in sequences. In *Proc. of the First Int'l Conference on Knowledge Discovery and Data Mining*, pages 210–215, Montreal, Quebec, 1995.

[SA96] R. Srikant and R. Agrawal. Mining sequential patterns: Generalizations and performance improvements. In *Proc. of the Fifth Int'l Conference on Extending Database Technology*, Avignon, France, 1996.

[Zak98] M.J. Zaki. Efficient enumeration of frequent sequences. In *7th International Conference on Information and Knowledge Management*, 1998.

[Zak99] Mohammed J. Zaki. Parallel sequence mining on smp machines. In *Workshop On Large-Scale Parallel KDD Systems (in conjunction 5th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 57–65, San Diego, CA, august 1999.